NEPTUNE II User Manual
Version 1.0
17 January 2010

Table of contents

Environment variables

Before installing NEPTUNE, you must:

☑ Check for the existence of the JAVA_HOME environment variable. If it does not exist, you must define it with a valid path to the current java development kit. NEPTUNE requires at least version 1.6 of the JDK in order to work correctly.

JRE is insufficient because metamodels are compiled and you need a java version including java compiler

☑ Add to your path an access to the java and javac commands. To access these executables, you must add the variable “%JAVA_HOME%\bin” (windows form) or “${JAVA_HOME}/bin” (Unix form) to the path.

Examples:

♦ *In unix using sh shell, you must declare:*

```
JAVA_HOME=/usr/local/jdk1_6
export JAVA_HOME
path=$path:$JAVA_HOME/bin
export path
```

♦ *In windows XP, you must:*

```
Parameters->system->advanced->Environment variables
Create a new variable for a user:
      JAVA_HOME    “C:\Program Files\Java\jdk1.6.0_13”
Modify the path variable:
      Add at the end of this path   “…;%JAVA_HOME%\bin”
```

How to load a model?

There are two options to load a model.

☑ The first one to select in the metamodel browser (see Figure 1, ❶) the metamodel to which the model is conform and to click on the 'Load Model' option. A dialog box opens (see Figure 3) where the "Name of Metamodel" drop-down list is deactivated. Click on the button "…" (see Figure 2, ❹) to open the File chooser dialog box and select the XMI file containing your model. You can modify the model name in the text field "Name of the model".

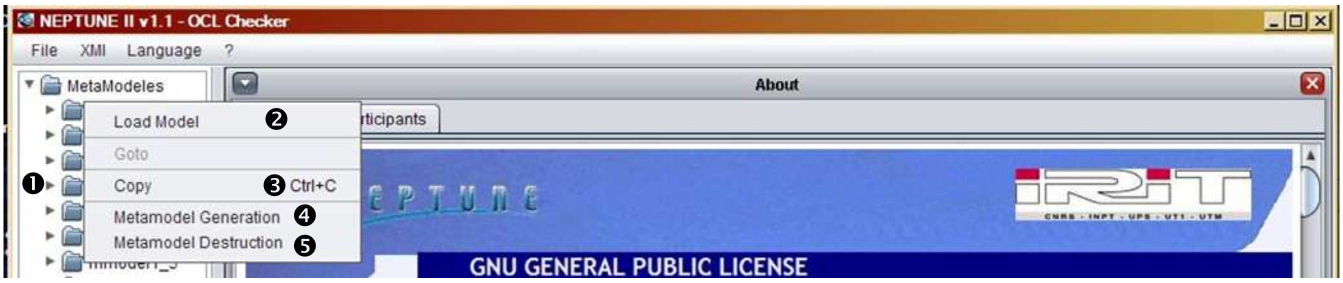
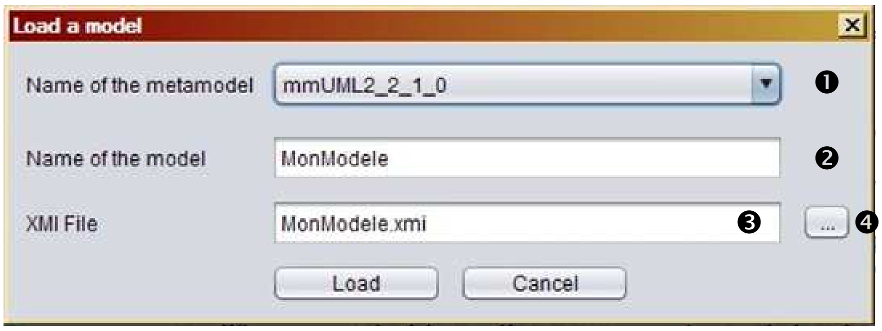


Figure 1: Metamodel browser and its menu



Name of the metamodel: mmUML2_2_1_0 ❶
Name of the model: MonModele ❷
XMI File: MonModele.xmi ❸ ... ❹

Figure 2: Load a model window

☑ The second one consists in selecting the "load Model" option in the XMI menu (see Figure 3, ❸). This opens the Load a model window displayed in Figure 2. In this case, the "Name of Metamodel" drop-down list is activated. You must select the metamodel corresponding to your model and click on the button "…" (see Figure 2, ❹) to open the File chooser dialog box then select the XMI file containing your model. You can modify the model name in the text field "Name of the model".

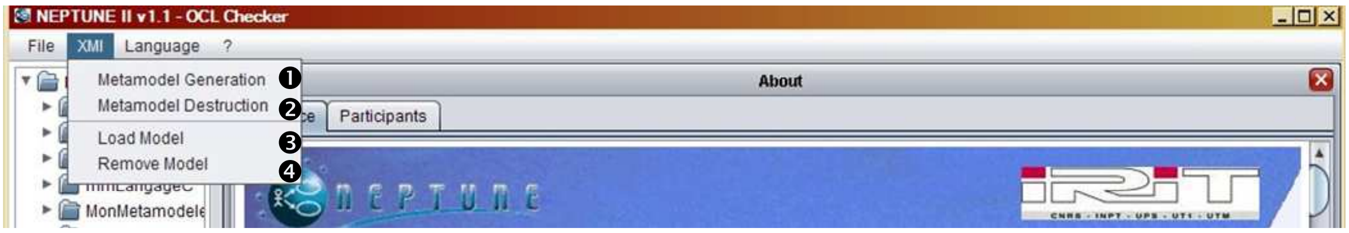


Figure 3: XMI menu

When a model is loaded, it appears in the model browser (Figure 4, ❶).

How to remove a model?

There are two options to remove a model.

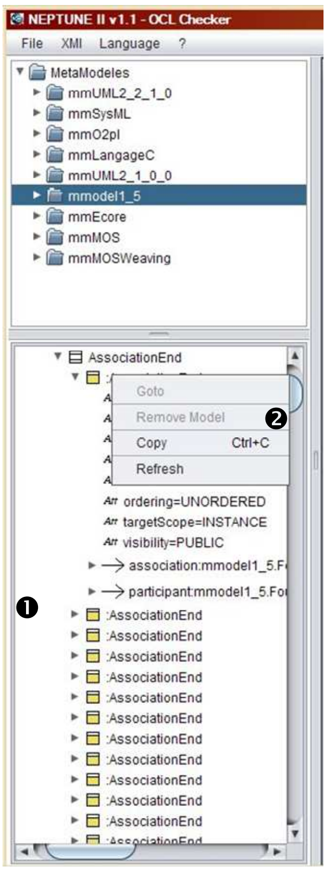


Figure 4: Model browser and its menu

- ☑ The first one consists to select in the model browser (see Figure 4, ❶) the model you want to remove and to click on the “Remove Model” option (see Figure 4, ❷). This opens the delete windows where the two text fields are deactivated. Clicking on the “Delete” button opens a confirmation box where you confirms the removal. After selecting “yes”, the model is removed.
- ☑ The second one consists in selecting the “Remove Model” option in the XMI menu (see Figure 3, ❹). This opens the remove model window displyed in Figure 5. In this case, the “Name of the Metamodel” and “Name of the model” drop-down lists are activated. You must first select the metamodel corresponding to your model then select the model in the list “Name of the model”. Finally, click on “*Delete*” to complete the deletion.

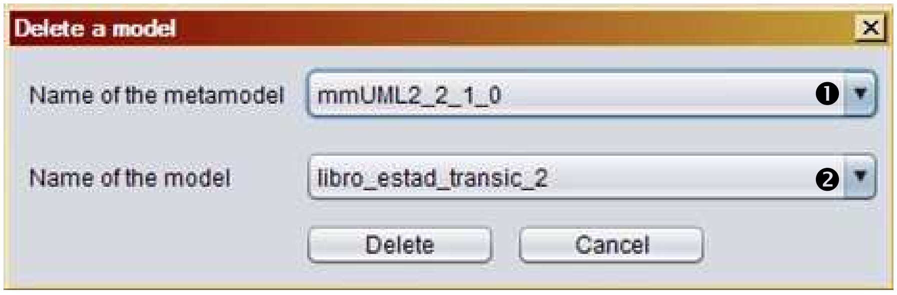
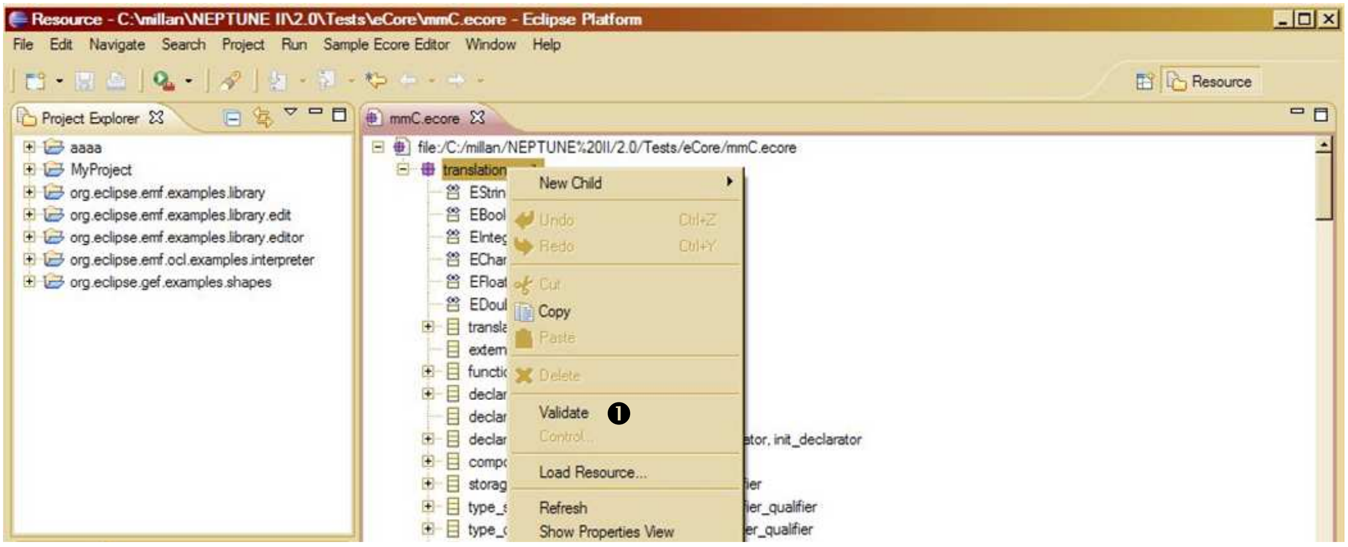


Figure 5: Remove model window

How to generate a metamodel?

Generating a metamodel is a very delicate operation. Indeed, it requires having the metamodel description in XMI 1.5 or in Ecore format. In addition, the metamodel must be validated in order to be well formed. If the metamodel is created using the Eclipse platform, you must use the “Validate” option as illustrated in Figure 6 (❶) before generating a metamodel.


Resource - C:\millan\NEPTUNE II\2.0\Tests\eCore\mmC.ecore - Eclipse Platform
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Project Explorer
aaaa
MyProject
org.eclipse.emf.examples.library
org.eclipse.emf.examples.library.edit
org.eclipse.emf.examples.library.editor
org.eclipse.emf.ocl.examples.interpreter
org.eclipse.gef.examples.shapes
mmC.ecore
file:/C:/millan/NEPTUNE%20II/2.0/Tests/eCore/mmC.ecore
translation
New Child
Undo Ctrl+Z
Redo Ctrl+Y
Cut
Copy
Paste
Delete
Validate ❶
Control...
Load Resource...
Refresh
Show Properties View
Resource
Figure 6: Eclipse menu to validate Ecore metamodels

There are two options to generate a metamodel.

☑ The first one consists to select in the metamodel browser (see Figure 1, ❶) the option “Metamodel Generation” (see Figure 1, ❹). This opens a dialog box (see Figure 7). Click on the button (see Figure 7, ❸) to open the File chooser dialog box and select the XMI file containing your metamodel. You can modify the metamodel name in the text field “Metamodel name”.

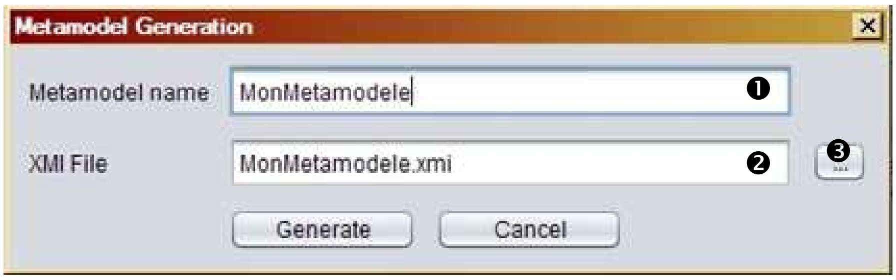
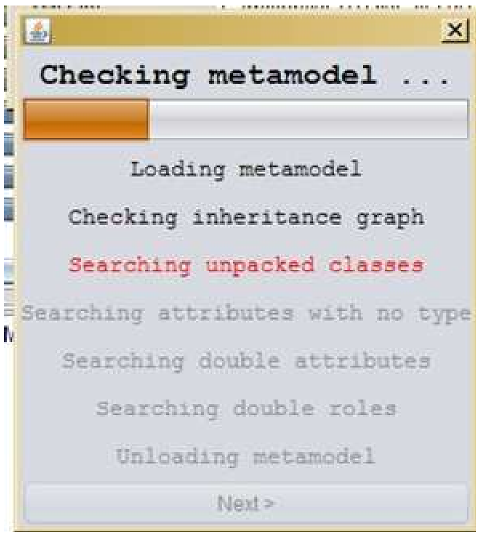


Figure 7: Metamodel generation window

☑ The second one consists in selecting the “Metamodel Generation” option in the XMI menu (see Figure 3, ❶). This opens the load metamodel generation window displayed in Figure 7. You must click on the button “…” (see Figure 7, ❸) to open the file chooser dialog box and select the XMI file containing your metamodel. You can modify the metamodel name in the text field “Metamodel name”.

A partial validation is also performed by NEPTUNE before generating a metamodel. Figure 8 shows the different steps of a metamodel validation. If no error is detected, the internal form of the metamodel is generated and compiled.



Checking metamodel ...

Loading metamodel

Checking inheritance graph

Searching unpacked classes

Searching attributes with no type

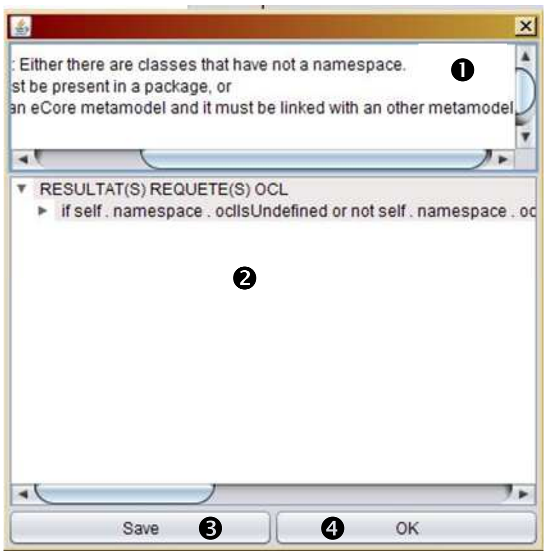
Searching double attributes

Searching double roles

Unloading metamodel

Figure 8: NEPTUNE metamodel validation

"Searching unpacked classes" is only a warning. When metamodels reference other metamodels, some classes are not in a namespace because this namespace is part of an other metamodel. If a warning or an error occurs, the box below (Figure 9) is displayed. This box displayed the problem type (error or warning), a description of the problem (❶) and the result of the test (❷) to find the element for which the test failed. "Save" allows to save the result on disk and "ok" allows to continue the validation.



: Either there are classes that have not a namespace.
st be present in a package, or
an eCore metamodel and it must be linked with an other metamodel

RESULTAT(S) REQUETE(S) OCL
if self . namespace . oclIsUndefined or not self . namespace . oc

Figure 9: NEPTUNE metamodel error or warning box

In order to avoid conflicts between the metamodel name and a namespace available in the metamodel, we recommend prefixing the metamodel name by "mm". For example, the name of the C language metamodel could be "mmLangageC".

The NEPTUNE metamodel validation tool is not able to detect all the errors in a metamodel. So the compiling of the metamodel can fail. It is important to carry out the results of the compilation and to correct the metamodel. **In this case, the partially compiled metamodel must be manually removed.**

Ecore metamodels can have references to metaelements from other metamodels. In this case, the box below (Figure 10) is displayed.

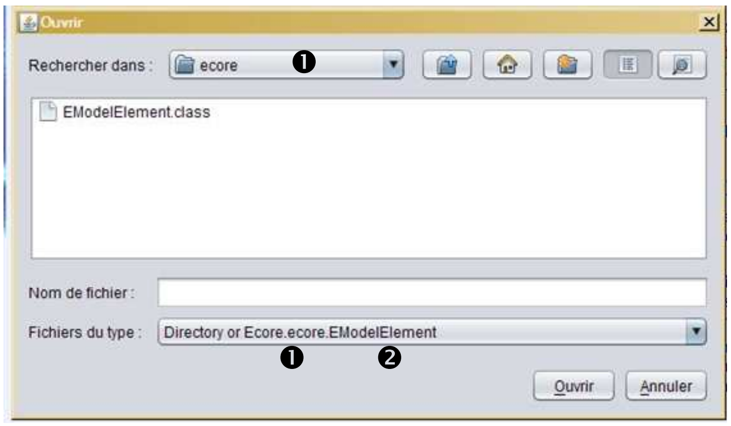


Figure 10: Metaclasses selector

This dialog box allows users to select the different elements required for generating a new metamodel. ❶ indicates the metamodel where the required metaclass is located and ❷ indicates the metaclass. If different versions of a same metamodel co-exist in NEPTUNE, the user must select the correct one.

How to delete a metamodel?

There are two options to delete a metamodel.

- ☑ The first one consists in selecting in the metamodel browser (see Figure 11, ❶) the metamodel you want to delete. This opens the “Metamodel Destruction” window where the text field is deactivated. Clicking on the “Destruction” button opens a dialog box asking for the confirmation of the destruction. After selecting “yes”, the metamodel is deleted.
- ☑ The second one consists in selecting the “Metamodel Destruction” option in the XMI menu (see Figure 3, ❷). This opens the “Metamodel Destruction” window displayed in Figure 11. In this case, the “Name of the Metamodel” drop-down list is activated. Select the metamodel you want to delete and click on the “*Destruction*” button.

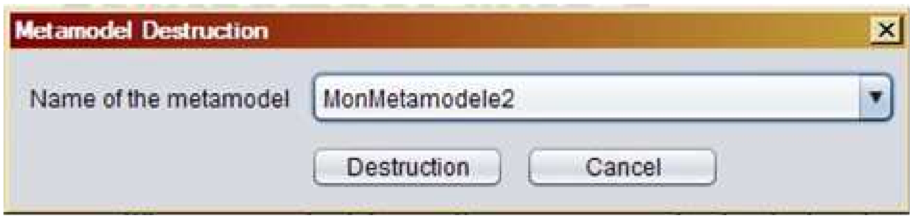
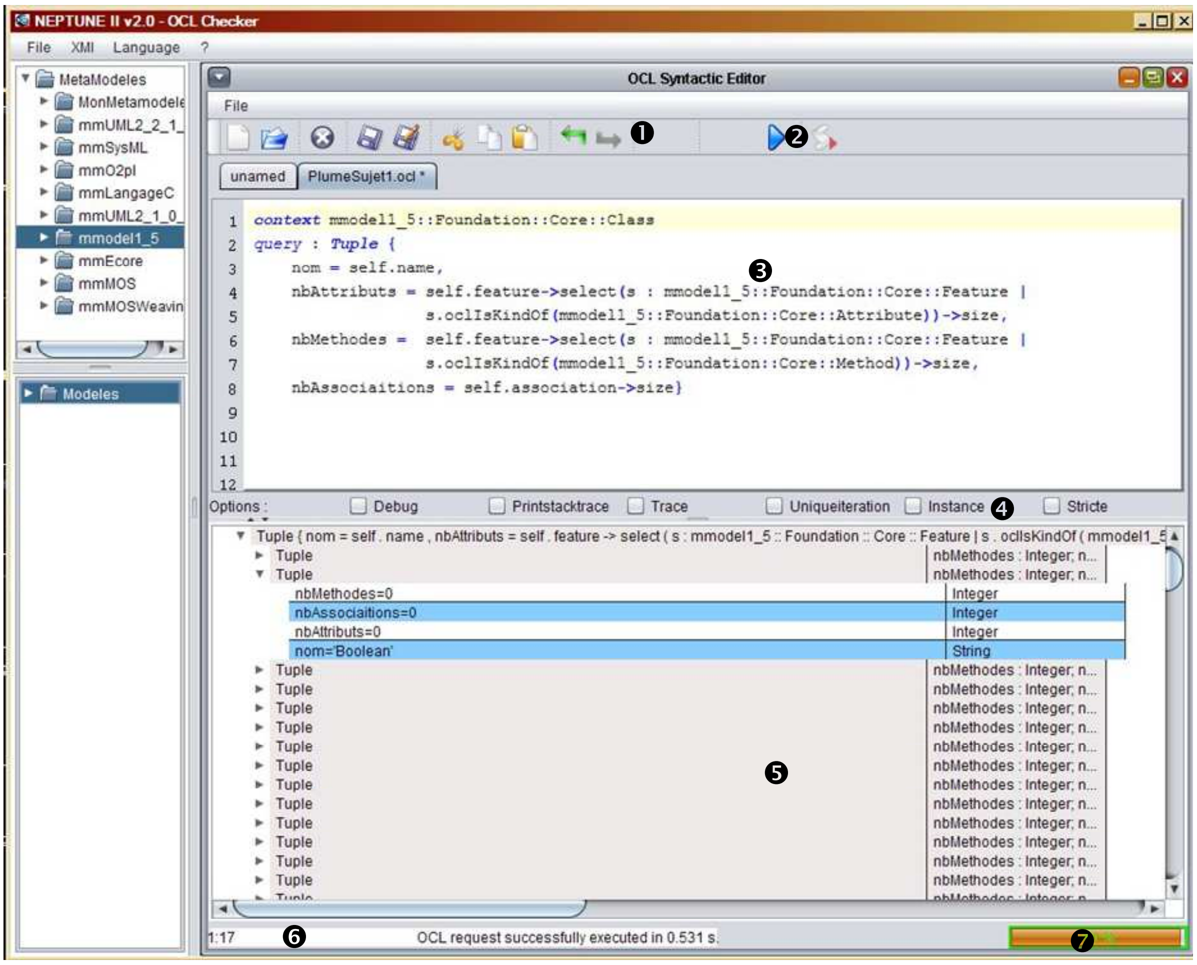


Figure 11: Delete metamodel window

In order to avoid problems, it is recommended to exit from NEPTUNE and to start it again.

Graphical User Interface Description



```
context mmodel1_5::Foundation::Core::Class
query : Tuple {
    nom = self.name,
    nbAttributs = self.feature->select(s : mmodel1_5::Foundation::Core::Feature |
                  s.oclIsKindOf(mmodel1_5::Foundation::Core::Attribute))->size,
    nbMethodes =  self.feature->select(s : mmodel1_5::Foundation::Core::Feature |
                  s.oclIsKindOf(mmodel1_5::Foundation::Core::Method))->size,
    nbAssociaitions = self.association->size}
```

```
▼ Tuple { nom = self . name , nbAttributs = self . feature -> select ( s : mmodel1_5 :: Foundation :: Core :: Feature | s . oclIsKindOf ( mmodel1_5
  ► Tuple                                    nbMethodes : Integer; n...
  ▼ Tuple                                    nbMethodes : Integer; n...
      nbMethodes=0                             Integer
      nbAssociaitions=0                        Integer
      nbAttributs=0                            Integer
      nom='Boolean'                            String
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...
  ► Tuple                                    nbMethodes : Integer; n...

1:17          OCL request successfully executed in 0.531 s.
```

Figure 12: Neptune User Interface

As shown n Figure 12, the NEPTUNE GUI consists of two areas. The first area on the left of the window contains two browsers allowing to navigate in metamodels and in models. The second area contains internal frames for editing and executing pOCL rules.

The browsers

Figure 4 shows the two browsers: the metamodel browser on the top the model browser on the bottom. Each of them uses icons showing the kind of elements handled.

Common icons for both metamodel and model browsers

⊟	A metaclass.
Att	A metaclass or object attribute.
⟶	An association between the metaclass and the specified metaclass.

Metamodel browser specific icons

An inheritance link between the metaclass and the specified metaclass.

An enumerative.

Model browser specific icons

An object instance of the metaclass in which the object appears.

An object part of an opposite association end of the considered object.

If no model is specified in an OCL rule, the model selected in the model browser is used.

The internal frames

Toolbars

❶ contains all the functionalities for creating, saving, editing pOCL rules.

Create a new tab to edit and to process pOCL rules.

Open a file containing a pOCL rule. Clicking on this button displays a dialog box where you select a rule. This dialog box is similar to those shown in Figure 10.

Close the active tab. If the rule currently edited is modified, a dialog box where you save the rule before you closing the tab.

Save the active tab. If it is the first save of the rule, a dialog box is displayed where you select directory to save the rule and you provide the name of the file used to store the rule.

Save the active tab. Clicking on this button always opens the dialog box where you select the directory to save the rule and you provide the name of the file used to store the rule.

Cut the selected part of the rule and save it in the clipboard.

Copy the selected part of the rule and save it in the clipboard.

Paste the text in the clipboard into the editor at the current cursor position

undo

redo

❷ contains the button to process a rule and the button to save the result of this runtime on disc.

▶	Process the rule currently edited
💾	Save the result of the runtime in text file. A dialog box opens where you select the repository where to save the rule and you provide the name of the file used to store the result is displayed.

⚠ For performance reason it is impossible to process several rules simultaneously. Indeed, processing several rules simultaneously requires using synchronous collections which increase the access time significantly. For huge model this increase is too prejudicial.

Tabs

A tab consists of four zones constitute a tab. The first one (❸) is the rule editor. This editor supports the syntactic coloration. OCL keywords are displayed in blue and bold italic style. pOCL specific keywords are displayed in blue and italic style. Strings are displayed in blue and comments in light gray. Four characters replace tabulation. The second area (❹) contains all the options of the pOCL interpreter. There are six options.

Debug	Display in the console information about the nodes crossed by the interpreter. ***This option is only used by expert to debug the interpreter.***
Printstacktrace	Display in the console the runtime stack when errors are generated by the interpreter. ***This option is only used by expert to debug the interpreter.***
Trace	Display in the console information for debugging NEPTUNE interpreter. In particular, this option displays intermediate results. ***This option is only used by expert to debug the interpreter.***
Uniqueiteration	This option indicates to the interpreter that the rule is only executed for the first instance processed. After this first iteration, the interpreter returns the result. This option can be used for example for rules beginning by “X.allInstances.”
Instance	This option indicates to the interpreter that the rule is only executed for the selected instance. The instance is selected in the model browser. After this iteration, the interpreter returns the result.
Strict	This option imposes to the interpreter to verify that each expression is completely and correctly typed. In particular, this option forbids use of implicit variables in iterate expressions.

The third area (❺) displays the result of the processed rule. Each gray line displays the exact type of the rule processed on an instance. This gray line is a node containing the value calculated by the interpreter. If the result is a tuple, each leaf corresponds to an attribute of the tuple. If the result is a collection, each leaf corresponds to a value contained in the collection. The attributes or the values are displayed alternatively in white and in blue.

The fourth area (❻) displays the current position of the cursor in the editor and the duration of the runtime once terminated.

❼ is common to all tabs. This progression bar indicates the amount of memory available for NEPTUNE. Clicking on it runs the garbage collector.

The menus

NEPTUNE menus

Four menus are available in the main menubar of the NEPTUNE application (see Figure 13).

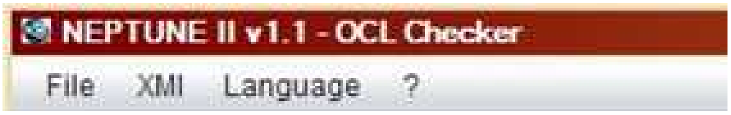


Figure 13: Neptune menubar

The first menu groups (Figure 14) the option to open editor and consoles.

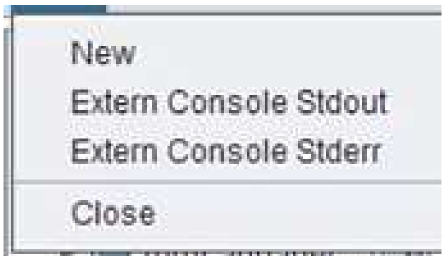


Figure 14: File menu

New	Open a new pOCL editor.
Extern Console Stdout	Open a console displaying the standard output messages.
Extern Console Stderr	Open a console displaying the error messages in red.
Close	Close the NEPTUNE tool. This action is similar to clicking on the top right cross.

The second menu (Figure 15) groups the options to handle models and metamodels.

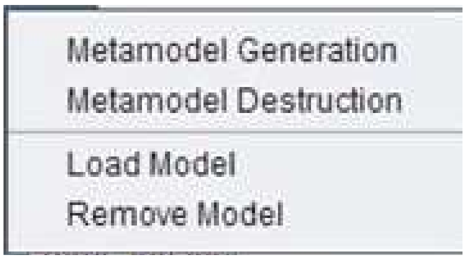


Figure 15: XMI menu

Metamodel Generation	This opens the load metamodel generation window displayed in Figure 7. You must click on the button "…" (see Figure 7, ❸) to open the file chooser dialog box and select the XMI file containing your metamodel. (See section "How to generate a metamodel?")
Metamodel Destruction	This opens the "Metamodel Destruction" window displayed in Figure 11. In this case, the "Name of the Metamodel" drop-down list is activated. Select the metamodel you want to delete and click on the "Destruction" button. (See section

	"How to delete a metamodel?")
Load Model	This opens the Load a model window displayed in Figure 2. In this case, the "Name of Metamodel" drop-down list is activated. You must select the metamodel corresponding to your model and click on the button "…" (see Figure 2, ❹) to open the File chooser dialog box then select the XMI file containing your model. (See section "How to load a model?")
Remove Model	This opens the remove model window displyed in Figure 5. In this case, the "Name of the Metamodel" and "Name of the model" drop-down lists are activated. You must first select the metamodel corresponding to your model then select the model in the list "Name of the model". (See section "How to remove a model?")

The third menu (Figure 16) allows to change the current language used by the NEPTUNE interface. Currently NEPTUNE supports French and English. By default, the English language is selected.

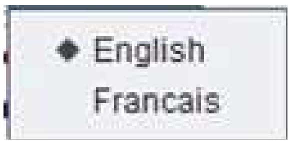


Figure 16: Language menu

The fourth menu (Figure 17) contains additional functions as "Help" which displays help information (not yet implemented) and "About" which displays NEPTUNE contributors and NEPTUNE licence.

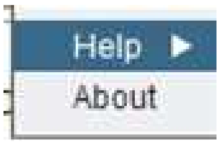


Figure 17: ? menu

Editor menu

Figure 18: pOCL Syntactic Editor menubar

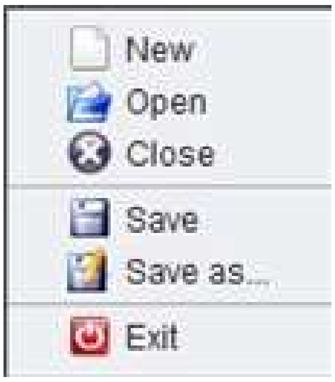


Figure 19: File pOCL editor menu

New	Create a new tab to edit and to process pOCL rules.
Open	Open a file containing a pOCL rule. Clicking on this button displays a dialog box where you select a rule. This dialog box is similar to those shown in Figure 10.
Close	Close the active tab. If the rule currently edited is modified, a dialog box where you

	save the rule before you closing the tab.
Save	Save the active tab. If it is the first save of the rule, a dialog box is displayed where you select directory to save the rule and you provide the name of the file used to store the rule.
Save as	Save the active tab. Clicking on this button always opens the dialog box where you select the directory to save the rule and you provide the name of the file used to store the rule.
Exit	Close the current pOCL syntactic editor.

Contextual menus

Metamodel browser menu

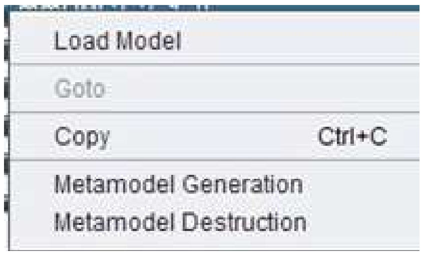
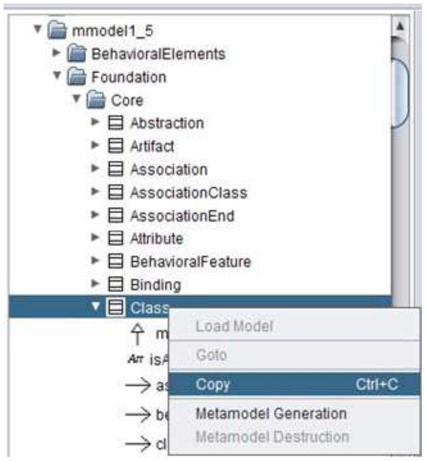


Figure 20: Metamodel contextual menu

Load Model	This opens the Load a model window displayed in Figure 2. In this case, the “Name of Metamodel” drop-down list is activated. You must select the metamodel corresponding to your model and click on the button “…” (see Figure 2, ❹) to open the File chooser dialog box then select the XMI file containing your model.
Goto	Goto allows to follow either an association end or an inheritance link. Click on “Goto” to access the description of the class in association with the selected class, or to the ancestor of the class if the considered link is an inheritance link.
Copy	Copy the path of the selected element in the browser.



Example 1: Example of the copy option

Example 1 shows an example of how to use the copy option. In this case the information in the clipboard is “*mmodel1_5::Foundation::Core::Class*”.

Metamodel Generation	This opens the load metamodel generation window displayed in Figure 7. You must click on the button "…" (see Figure 7, ❸) to open the file chooser dialog box and select the XMI file containing your metamodel.
Metamodel Destruction	This opens the "Metamodel Destruction" window displayed in Figure 11. In this case, the "Name of the Metamodel" drop-down list is activated. Select the metamodel you want to delete and click on the "Destruction" button.

Model browser menu

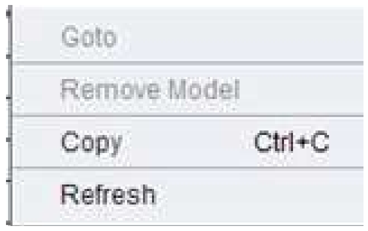


Figure 21: Model contextual menu

Goto	Goto allows to follow an association. Click on "Goto" to access the object in association with the selected objet.
Remove model	This opens the remove model window displyed in Figure 5. In this case, the "Name of the Metamodel" and "Name of the model" drop-down lists are activated. You must first select the metamodel corresponding to your model then select the model in the list "Name of the model".
Copy	Copy the path of the selected element in the browser.
Refresh	Refresh allows to display the models not displayed in the model browser. This is useful when a user create a new model during a transformation.

Editor menu

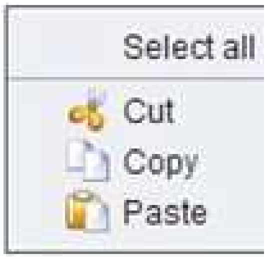


Figure 22: pOCL editor contextual menu

Select all	Select all the text in the pOCL editor.
Cut	Cut the selected part of the rule and save it in the clipboard.
Copy	Copy the selected part of the rule and save it in the clipboard.
Paste	Paste the text in the clipboard into the editor at the current cursor position.

pOcl extensions

Extension to write multimodel OCL rules

The main extension of OCL is the capability to handle simultaneously several models conform to different metamodels.

To indicate a metamodel in an OCL rule, you must prefix the path with the metamodel name.

```
package  mmO2pl::o2pl                                                  ❶

context mmO2pl::o2pl::Affectation_Contenu                              ❷
def : let instruction : Sequence(OclAny) =
      Sequence{self.affContSuiv, self.paramCont, self.valCont}

def : let registre : Bag(String) =
      mmMos[atv_bds1, atv_bds2]::mos::RootDefinition::allInstances.     ❸
            rechercheRegistre('ccdTemp')->flatten

def : estTerminal(cfs : Bag(mmSysML::uml::ControlFlow)) : Boolean =
      cfs.target->select(an : mmSysML::uml::ActivityNode |
         an.oclIsKindOf(mmSysML::uml::ActivityFinalNode))->size = cfs->size  ❹
```

Example 2: metamodel and model naming in an OCL rule

Example 2 shows three use cases of the metamodel name. In ❶, the metamodel name is used in a package definition in order to specify the metamodel where to find the package "*o2pl*" for example. In ❷, the metamodel name is used in a context definition in order to specify the metamodel where to find the class for example "*o2pl::Affectation_Contenu*". In ❸ and ❹, the metamodel name is used in an expression definition for specifying the complete path to a specific class for example "*mmMos[atv_bds]::mos::RootDefinition*" or "*mmSysML::uml::ActivityFinalNode*".

To specify models in OCL rule, you must add the model between square brackets separated by commas. When several models are specified, the rule is executed on all models. In ❸, "*allInstances*" is applied to all instances of the class "*mos::RootDefinition*" in the models "*atv_bds1*" and "*atv_bds2*".

Definition of functions on OCL datatypes

NEPTUNE supports definition of new operations on OCL datatypes. Example 3 illustrates the operation "*startsWith*" that returns true if "*self*" starts with the string "*s*".

```
context String
def: startsWith(s : String) : Boolean =
      s.size() <= self.size() and self.substring(1, s.size()) = s
```

Example 3: operation definition on OCL datatype

Syntax extension

A problem occurs when a model element has the same name as an OCL keyword.

```
<eClassifiers xsi:type="ecore:EClass" name="Comment" eSuperTypes="#//Element">
      …
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="body" ordered="false"
            eType="#//String" unsettable="true">

      …
      </eStructuralFeatures>
</eClassifiers>
```

Example 4: XMI fragment of a metamodel

In Example 4, a fragment of a metamodel defines a classifier with an attribute called "*body*". "*body*" is an OCL keyword defining the operation body.

```
context mmSysML[setstartrackertovacuumtemperaturecomp]::uml::DecisionNode
def : estTemperature(cfs : Bag(mmSysML::uml::ControlFlow),  cond : String) : Boolean
            = cfs.guard->select(c : mmSysML::uml::ValueSpecification |
                  c.oclIsKindOf(mmSysML::uml::OpaqueExpression)).
                        body->flatten.contains(cond)->includes(true)
```

Example 5: OCL rule using keyword as model element name – version 1

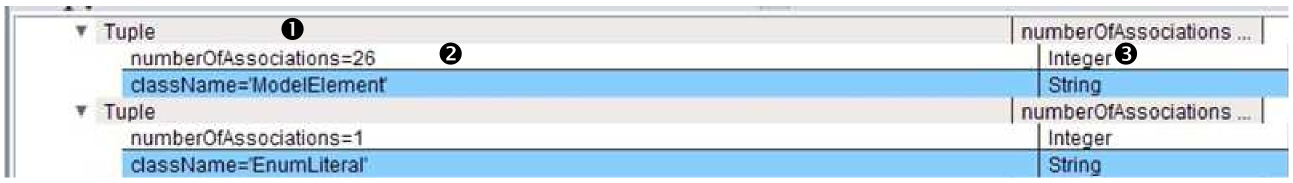
Example 5 is an OCL definition using "*body*". In this operation definition, the use of the metaelement "*body*" conflicts with the OCL keyword "*body*". In order to avoid such a conflict, metaelements having the same name as an OCL keyword must be surrounded with underscores as in Example 6.

```
context mmSysML[setstartrackertovacuumtemperaturecomp]::uml::DecisionNode
def : estTemperature(cfs : Bag(mmSysML::uml::ControlFlow),  cond : String) : Boolean
            = cfs.guard->select(c : mmSysML::uml::ValueSpecification |
                  c.oclIsKindOf(mmSysML::uml::OpaqueExpression)).
                        _body_->flatten.contains(cond)->includes(true)
```

Example 6: OCL rule using keyword as model element name – version 2

Three new OCL keywords have been added to NEPTUNE. In the pOCL editor, these keywords are in blue and in italic. "*query*" allows to define queries on a model. It is then possible to define views and metrics. All the results returned by a query have been correctly typed and this type is part of the returned result.

```
context mmodel1_5[umlmodel15]::Foundation::Core::Class
query : Tuple {   className : String = self.name,
                  numberOfAssociations : Integer = self.association->size}
```



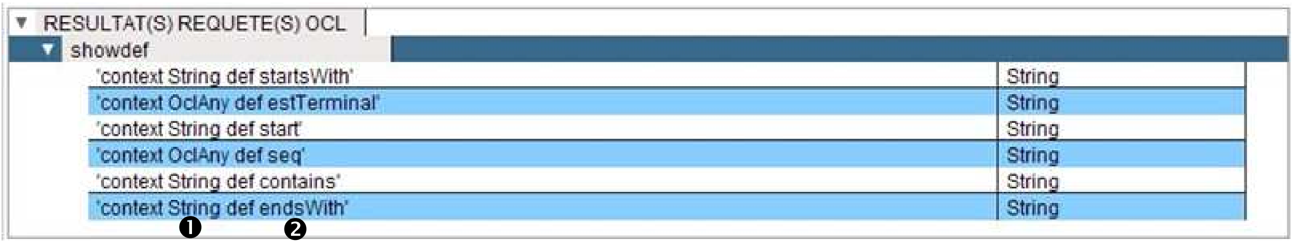
```
▼ Tuple ❶                                  | numberOfAssociations ...
    numberOfAssociations=26 ❷              | Integer ❸
    className='ModelElement'               | String
▼ Tuple                                    | numberOfAssociations ...
    numberOfAssociations=1                 | Integer
    className='EnumLiteral'                | String
```

Example 7: pOCL rule using a query

Example 7 shows a rule allowing to extract the name of each class and the number of its associations. The result of this query contains an indication of the data structure used (❶), the elements composing this structure (❷) and the type of each element (❸).

"*showdef*" displays all the operations defined. The context is only required to have an OCL rule syntactically well formed. Example 8 shows the additional operation defined to write rules. An operation is defined on a context (❶) and has a name (❷).

```
context mm02pl
showdef
```



```
▼ RESULTAT(S) REQUETE(S) OCL
  ▼ showdef
      'context String def startsWith'      | String
      'context OclAny def estTerminal'     | String
      'context String def start'           | String
      'context OclAny def seq'             | String
      'context String def contains'        | String
      'context String def endsWith'        | String
                 ❶         ❷
```

Example 8: showdef command

An operation can be used on an element having the same context as the operation or on an element having a type compatible with the context on which the operation is defined. For example, an operation defined on real is usable on integer. Operations defined in NEPTUNE support overloading and dynamic binding.

"*undef*" removes an operation defined in a context.

```
context String
undef : startsWith
```

Example 9: undef command

Example 9 illustrates a rule to remove the "*startsWith*" defined on the context "*String*".

Library extensions

The OCL library is compliant with the OCL 2.0 standard. However, the NEPTUNE OCL library for primary types and collections includes all the operations specified in the "Object Constraint Language – OMG Available Specification – Version 2.1 (ptc/09-05-02)". In order to use NEPTUNE for different projects, we add several new operations on "String" and "Collection".

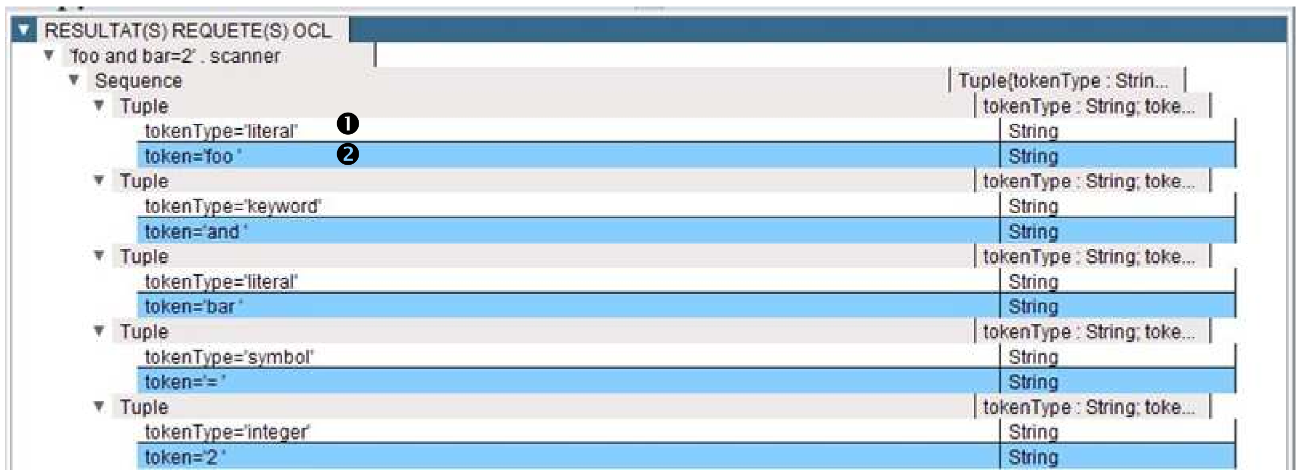
OclVoid

"oclUndefined" operator returns an oclUndefined object. This operation is mainly used in transformation in order to provide a result when an assignment or a creation failed (Example 13, ❺).

String

In order to analyse conditional expression used in decision nodes for example, the "*String*" library includes an operation called "*scanner*" returning the sequence of tokens present in the expression where the scanner operation is applied. This operation returns a sequence of tuple made of an attribute containing the token (see Example 10, ❷) and another containing the type of the token: "*keyword*", "*literal*", "*symbol*", "*integer*", "*boolean*", "*string*" and "*real*" (see Example 10, ❶).

```
'foo and bar=2'.scanner
```



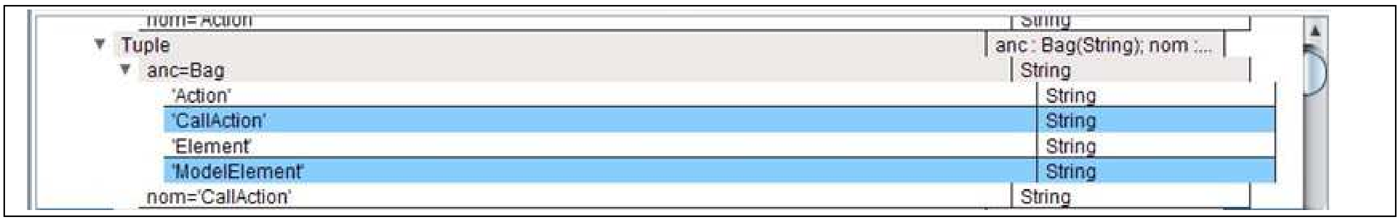
```
RESULTAT(S) REQUETE(S) OCL
  'foo and bar=2' . scanner
    Sequence                                  Tuple{tokenType : Strin...
      Tuple                                   tokenType : String; toke...
        tokenType='literal'    ❶                String
        token='foo '           ❷                String
      Tuple                                   tokenType : String; toke...
        tokenType='keyword'                     String
        token='and '                            String
      Tuple                                   tokenType : String; toke...
        tokenType='literal'                     String
        token='bar '                            String
      Tuple                                   tokenType : String; toke...
        tokenType='symbol'                      String
        token='= '                              String
      Tuple                                   tokenType : String; toke...
        tokenType='integer'                     String
        token='2 '                              String
```

Example 10: scanner operation

Collection

A model can be considered as an oriented graph. In this context, it could be useful to calculate the whole path between two nodes of a model. "*closure*" returns a collection resulting of the application of an OCL expression on the elements of a collection recursively until no new calculated element can be added. This operation implements a fixed point algorithm.

```
context mmodel1_5[umlmodel15]::Foundation::Core::Class
query : Tuple{nom = self.name, anc = self->closure(c :
      mmodel1_5::Foundation::Core::Class | c. c.generalization.parent).name}
```



```
nom='Action'                     String
▼ Tuple                          anc : Bag(String); nom :...
  ▼ anc=Bag                      String
      'Action'                   String
      'CallAction'               String
      'Element'                  String
      'ModelElement'             String
    nom='CallAction'             String
```

Example 11: closure operation

Example 11 shows a rule allowing to complete all the ancestors of "*self*". The result of this computing includes "*self*".

Dynamic typing

The NEPTUNE pOcl interpreter supports dynamic typing of expressions. For each evaluated element the correct type is automatically computed. In this context, the OCL operator "*OclAsType*" is now useless because the dynamic type is always computed during an evaluation.

```
context mmodel1_5[umlmodel15]::Foundation::Core::Classifier                   ❶
query :  mmodel1_5::Foundation::Core::Classifier::allInstances

context mmodel1_5[umlmodel15]::Foundation::Core::Classifier
query :  mmodel1_5::Foundation::Core::Classifier::allInstances->              ❷
      select(cl | cl.oclIsKindOf(mmodel1_5::Foundation::Core::Class))

context mmodel1_5[umlmodel15]::Foundation::Core::Classifier
query :  mmodel1_5::Foundation::Core::Classifier::allInstances->              ❸
      select(cl | cl.oclIsKindOf(mmodel1_5::Foundation::Core::Class)).isActive
```

Example 12: dynamic typing

Example 12 shows three pOCL rules. Evaluation (❶) returns as a set of "*Classifier*" while evaluatin (❷) returns a set of "*Class*". In this case, the evaluation of the rule shown in (❸) is valid although the "*oclAsType*" operation is not used.

pOcl extension for models transformation

An OCL extension allows side effects in order to transform models. This extension provides four operators and modifies the semantic of the OCL sequence.

pOcl operators

"*model*" sets up the default model on which the rules will be evaluated. To create a new model, it is necessary to use a "model" operation followed by "*new*" and the metamodel and model name (Example 13 - ❶).

Syntax: ***model new*** MetamodelName***[***ModelName***]***

⚠	The created model does not appear in the browser model (Figure 4). To display it, it is necessary to use the "refresh" item of the model browser contextual menu (Figure 21).

"*new*" creates a new instance of the mentioned metaclass. This new instance is created in the model stated in the metaclass path or in the default model (Example 13 - ❷).

"<-" is the assignment operator. The behaviour of this operator depends on the left member type. If the type is a data type or a metaclass, this operator affects the result of the right expression in the left member (Example 13 - ❸). If the type is a collection this operator adds the result of the right expression in the collection specified in the left member (Example 13 - ❻). The evaluation of the assignment operator returns a boolean: true if the assignment is correctly processed and false otherwise.

"*transform*" runs the transformation. All assignments and creations are performed.

⚠	If a rule contains assignments or creations without using the "*transform*" operator, an exception is generated indicating an inconsistency in the rule definition.

```
model new mmodel1_5[modelClass]                                                    ❶


context mmodel1_5[libro]::BehavioralElements::StateMachines::SimpleState
def : State2Class : Sequence(Boolean) =
  Sequence {
    class <- new mmodel1_5[modelClass]::Foundation::Core::Class,                   ❷
    class.name <- self.name,                                                       ❸
    class.generalization <- self.Generalization(class)
   }

context mmodel1_5[libro]::BehavioralElements::StateMachines::SimpleState
def : ClassPere : mmodel1_5[modelClass]::Foundation::Core::Class =
  if Sequence {                                                                    ❹
    class <- new mmodel1_5[modelClass]::Foundation::Core::Class,
    class.name <- 'State'}->excludes(false) then
      class
  else
     self.oclUndefined                                                             ❺
  endif
```

```
context mmodel1_5[libro]::BehavioralElements::StateMachines::SimpleState
def : Generalization(fils: mmodel1_5[modelClass]::Foundation::Core::Class) :
Sequence(Boolean) =
  Sequence {
    gen <- new mmodel1_5[modelClass]::Foundation::Core::Generalization,
    if  mmodel1_5[modelClass]::Foundation::Core::Class::allInstances
       -> exists(c: mmodel1_5[modelClass]::Foundation::Core::Class | c.name='State')
    then
      gen.parent <- mmodel1_5[modelClass]::Foundation::Core::Class::allInstances
       -> select(c: mmodel1_5[modelClass]::Foundation::Core::Class | c.name='State')
    else
      gen.parent <- self.ClassPere                                                  ❻
    endif,
    gen.child <- fils
  }

context mmodel1_5[libro]::BehavioralElements::StateMachines::SimpleState
transform : self.State2Class                                                        ❼
```

Example 13: transformation example

Example 13 illustrated a transformation consisting in generating a class diagram from a state-transition diagram using the design pattern state. This example is focused on the creation of one class by state having a common class ancestor called "*State*". The source model is called "*libro*" and the target model "*modelClass*".

⚠	It is possible to make on place transformation. However, the lack of an operator removing an element of a collection limits this kind of transformation. So on place transformation must be limited to punctually adding or modifying a property. If the transformation is more complex, it is necessary to create a new model.

⚠	Line ❺ in Example 13 will return an "*oclUndefined*" object. "*oclUndefined*" is an instance of the bottom type "*OclVoid*" that can be substituted to all type and *metaclasse* used.

Sequence semantic

To perform a transformation, the semantic of the sequence collection has been modified in order to force the evaluation of the sequence's elements from the left to the right. This addition to the sequence semantic does not interfere with the initial semantic because no information about the evaluation of sequence's elements is described in the OCL standard.

pOcl extension for model serialisation

As NEPTUNE delivers tools to transform models, it is also necessary to provide functionalities for model serialisation. In this version, two serialisations are provided: text serialisation and XMI serialisation. To support serialisation, the "OclVoid" and "Collection" have been enriched with new operations. Three are required in order to serialise models in XMI format and two for serialisation in Text format.

⚠	This extension is experimental and no test has been performed yet.

XMI format

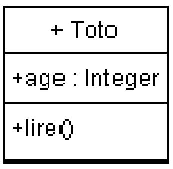
"OclVoid_int outputXMI(OclVoid_int father, Bag_int<OclVoid_int> attributs)": This operation creates a new XMI mark having the type of the context of the rule as name. "*father*" provides either the name of the file if no name has been specified (❶), or the name of the mark (❸). "*attributs*" represents the mark's attributes. The attribute "*xmi.id*" is automatically generated. This operation returns the element on which the rule is applied.

"*OclVoid_int outputXMI(OclVoid_int grandfather, OclVoid_int father, Bag_int<OclVoid_int> attributs)*": This operation is similar to the previous one but references both the grandfather and the father of the mark. The "*xmi.idref*" is derived from the object invoking the operation. In (❹), "*xmi.idref*" references the "*xmi.id*" calculated from "*self.child*". This operation returns the element on which the rule is applied.

"*Collection_int<T> outputRoleXMI(OclVoid_int pere, OclVoid_int role)*": This operation creates attributes for association ends. This operation is only applicable on collections containing the element referenced by the role. This operation returns the elements on which the rule is applied.

```
context ModelManagement::Model
query : self.outputXMI( 'apres.xmi',Bag{'xmi.id','isLeaf','name','isSpecification',
                        'isRoot','isAbstract'})                                  ❶

context ModelManagement::Model
query : Bag{self}->outputRoleXMI(self,'Namespace.ownedElement')                  ❷

context Foundation::Core::Class
query : self.outputXMI('Namespace.ownedElement',
            Bag{'xmi.id','name','visibility','isActive','isLeaf','isRoot'})      ❸

context Foundation::Core::Generalization
query :self.outputXMI('Namespace.ownedElement',Bag{'xmi.id','isSpecification'})

context Foundation::Core::Generalization
query : Bag {self}->outputRoleXMI(self,'Generalization.child')

context :Foundation::Core::Generalization
query : Bag {self}->outputRoleXMI(self,'Generalization.parent')

context Foundation::Core::Generalization
query : self.child.outputXMI(self,'Generalization.child', Bag{'xmi.idref'})      ❹

context Foundation::Core::Generalization
query : self.parent.outputXMI(self,'Generalization.parent',Bag{'xmi.idref'})
```

Example 14: serialisation rule for class diagram in XMI format



```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmlns="UML" xmi.version="1.2" timestamp="Sat Nov 28 22:13:50 CET 2009"
xmlns:UML="org.omg.xmi.namespace.UML">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>NEPTUNE v1.1</XMI.exporter>
      <XMI.exporterVersion>
        0.24(5) revised on $Date: 2006-11-06 19:55:22 +0100 (Mon, 06 Nov 2006) $
      </XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
  </XMI.header>

  <XMI.content>
    <UML:Model xmi.id="neptune.metamodeles.mmodel1_5.ModelManagement.Model@9b6bcd"
          isLeaf="false" name="untitledModel" isRoot="false" isAbstract="false">
      <UML:Namespace.ownedElement>
        <UML:Class
          xmi.id="neptune.metamodeles.mmodel1_5.Foundation.Core.Class@633c70"
          name="Toto" isActive="false" isLeaf="false" isRoot="false"/>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```

Example 15: example of a serialisation using rules defined in Example 14

Example 15 shows an execution of the rules of the Example 14 on the trivial class diagram containing only one class: “Toto”. The result of this execution can be loaded in the NEPTUNE II platform.

Text format

“OclVoid_int outputText(OclVoid_int file, OclVoid_int text)”: this operation creates a new text file whose name is contained in “*file*” and writes the text “*text*” in it. This operation must be invoked first. It returns the element on which the rule is applied.

“OclVoid_int outputText(OclVoid_int texte)”: this operation writes the text “*text*” in the file previously created using the operation “*outputText (OclVoid_int file, OclVoid_int text)”*. This operation returns the elements on which the rule is applied.

⚠	When serializing in text mode, the first operation used must always be “*outputText(OclVoid_int file, OclVoid_int text)”*in order to create the file “*file*”.

```
context Foundation::Core::Attribute
def : construireAtt : String =
            'private ' + self.type.name + ' ' + self.name + ' ;\n'

def : construireSetAtt : String =
        '\tpublic void set' + self.name.firstToUppercase +'(' + self.type.name +
        ' val) \n' +
        '\t{ \n\t this.'+ self.name + ' = val ;\n\t} \n'

def : construireGetAtt : String =
        '\tpublic ' + self.type.name + ' set' + self.name.firstToUppercase +'() \n' +
        '\t{ \n\t return this.'+ self.name + ' ;\n\t} \n'

context Foundation::Core::Class
query : self.outputText('toto.txt',
                        'public class ' + self.name + '\n' +
                        '{' +
                         '\t' + self.feature->select(f |
                                   f.oclIsKindOf(Foundation::Core::Attribute))->

                                  iterate(att : Foundation::Core::Attribute;
                                          str : String = '' |
                                                str + att.construireAtt + '\n' +
                                                 att.construireSetAtt + '\n' +
                                                att.construireGetAtt) +'}')
```

Example 16: serialisation rule for class diagram in text format

```
public class Toto
{     private Integer age ;

      public void setAge(Integer val)

      {
       this.age = val ;
      }

      public Integer setAge()
      {
       return this.age ;
      }
}
```

Example 17: example of a serialisation using rules defined Example 16

Example 17 shows the result of the rule defined in Example 16 performed on the class diagram of the Example 14. This rule generates a java class schema from the UML class diagram.

⚠	Others components can be implemented for serialisation of models in different formats: java, C++, Ada…

Conformance

	OCL-MOF subset	**Full OCL**
Syntax	Yes	Yes
XMI	Not supported	Not supported
Evaluation		
• allInstances	Yes	Yes
• @pre postconditions	Only syntax is checked	Only syntax is checked
• OclMessage	Not supported	Not supported
• Navigating non-navigable associations[1]	No	No
• Accessing private and protected features	Yes	Yes

"The UML 2.0 Infrastructure and the MOF 2.0 Core specifications that were developed in parallel with this OCL 2.0 specification share a common core. The OCL specification contains a well-defined and named subset of OCL that is defined purely based on the common core of UML and MOF. This allows this subset of OCL to be used with both the MOF and the UML, while the full specification can be used with the UML only.

The following compliance points are distinguished for both parts.

1. Syntax compliance. The tool can read and write OCL expressions in accordance with the grammar, including validating its type conformance and conformance of well-formedness rules against a model.

2. XMI compliance. The tool can exchange OCL expressions using XMI.

3. Evaluation compliance. The tool evaluates OCL expressions in accordance with the semantics chapter. The following additional compliance points are optional for OCL evaluators, as they are dependent on the technical platform on which they are evaluated.

• allInstances()

• pre-values and oclIsNew() in postconditions

• OclMessage

• navigating across non-navigable associations

• accessing private and protected features of an object

The following table shows the possible compliance points. Each tool is expected to fill in this table to specify which compliance points are supported." [OCL06]

[1] Several tries have been realized in order to navigate non-navigable associations. We discard this feature due to the number of ambiguities generated by the lack of name of the association-end no-navigable. Generating random names does not seem to be a useful solution.

Bibliography

[OCL06] Object Constraint Language - OMG Available Specification - Version 2.0 - formal/06-05-01 May 2006 - http://www.omg.org/spec/OCL/2.0/PDF

[OCL09] Object Constraint Language (Beta 2) - OMG Available Specification - Version 2.1 - ptc/09-05-02 May 2009 - http://www.omg.org/spec/OCL/2.1/Beta2/PDF

Version notes

Version	***Date***	***Author***	***Comments***
V 1	18/11/2009	Thierry Millan	Initial version

Illustration table

Example table